Model-Based Quality Assurance of Windows Protocol Documentation

Wolfgang Grieskamp, Dave MacDonald, Nicolas Kicillof,
Alok Nandan, Keith Stobie, Fred Wurden
Microsoft Corporation

Abstract

Microsoft is producing high-quality documentation for Windows client-server and server-server protocols. Our group in the Windows organization is responsible for verifying the documentation to ensure it is of the highest quality. We are applying various test-driven methods including, when appropriate, a model-based approach. This paper describes certain aspects of the quality assurance process we put in place, and specifically focuses on model-based testing (MBT). Our experiences so far confirm that MBT ***works*** *and that it* ***scales****, provided it is accompanied by sound tool support and clear methodological guidance.*

1. Introduction

Protocol documentation is an important part of Microsoft's compliance obligations with the US Department of Justice and the European Union. Microsoft is committed to produce high-quality documentation for certain Windows client-server and server-server protocols. These documents enable licensees to interoperate with Microsoft's operating systems.

Our group in the Windows organization acts as the last instance inside of Microsoft in the quality assurance process for this documentation. Around 25,000 pages of documentation for over 200 protocols have to be thoroughly verified to ensure that they are completely *accurate*, so that licensees can implement protocols from it (and some previous domain knowledge).

This herculean effort requires the application of innovative methods and tools. We have devised a methodology called the *protocol quality assurance process* (PQAP), with model-based testing (MBT) as one of its cornerstones. Model-based testing has a decade-long tradition inside of Microsoft. The first tools supporting it were deployed at the end of the nineties. Various tools exist today, among those the product family called Spec Explorer [1][2][3], which was originally developed at Microsoft Research, and today is productized internally as part of our effort.

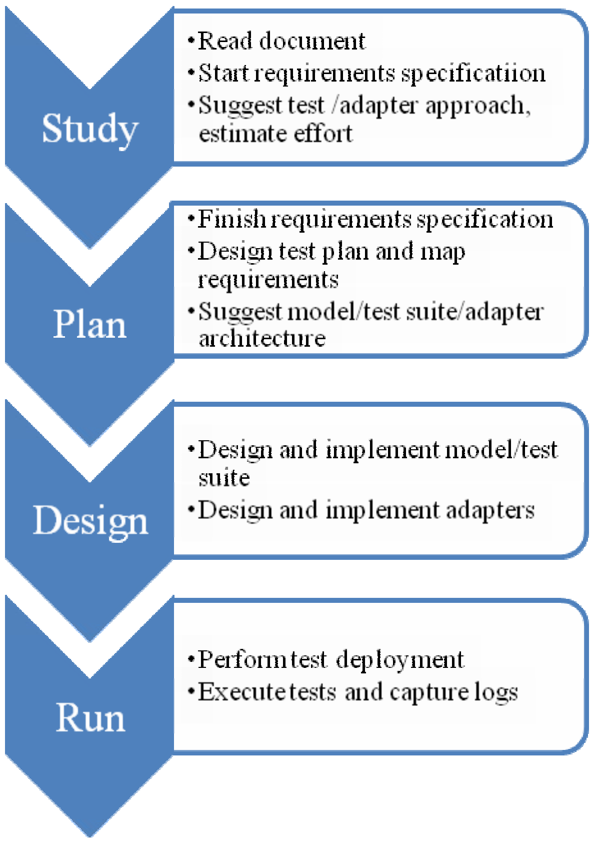
Though MBT has been applied successfully to features and products before [4][1], this is the first attempt to use it in such a large scale and in the context of a business-critical area within Microsoft, and to the best of our knowledge throughout the whole industry. Our experience so far confirms that MBT *works* and that it *scales*, provided it is accompanied by good tool support and clear methodological guidance, and backed up by investment in training.

2. The Protocol Quality Assurance Process

The basic idea behind the PQAP is *test-driven analysis of the documents*. By deriving a functional test-suite from a specification, we can ensure that it is thoroughly studied and that its normative statements are converted into assertions to be checked against the actual implementation. This formalization process naturally also validates internal consistency. It is furthermore close to the actual task of writing an implementation from the document, simulating a potential licensee's attempts.

All our test suite development in this effort is being performed by vendors in India and China. Employing vendors has a significant methodological dimension, since it establishes a clean-room environment which excludes interference of knowledge Microsoft employees may have about protocol implementations. Employing the large number of vendors required for the task, however, also causes significant challenges for the *management* and *control* of the test suite work itself. This is one of the major motivations for putting the PQAP in place.

Figure 1: Phases of the PQAP


Study
•Read document
•Start requirements specificatiion
•Suggest test /adapter approach, estimate effort
Plan
•Finish requirements specification
•Design test plan and map requirements
•Suggest model/test suite/adapter architecture
Design
•Design and implement model/test suite
•Design and implement adapters
Run
•Perform test deployment
•Execute tests and capture logs

2.1. Phases of PQAP

The PQAP is phase-oriented, guiding the development of a test suite for a given protocol through all the required steps. Its phases are described in Figure 1.
Every phase is followed by a formal review, with particular emphasis on the second one (Plan) and the last one (Run). As part of Plan Review, reviewers sign-off the approach for the given protocol, and as part of Run Review, the final result.
The progress and results of the PAQP are reflected in the PQAR (protocol quality assurance report), a single document to report on all phases of the process. Test suite developers fill in the PQAR incrementally. It is used as a basis for the reviews, and it constitutes the major artifact documenting the test suite. The PQAR is a template-based document with a fixed structure and a number of accompanying guidance documents for the various steps, stages, and phases.

2.2. Requirements Specification

While the PQAR is a central document to the process, the *requirements specification* (RS), mentioned in Figure 1, is equally important. This document is a table derived by test suite authors from the protocol documentation containing one entry for each explicit or implicit normative statement, which defines a unique identifier for the requirement that is referred to from other artifacts in the process. The RS specifies a description (linked to the original document) and other properties used to classify the requirement: whether it specifies client or server behavior, whether it is testable, and whether it will be verified in the test suite (and, if not, why). Test plans given in the PQAR refer to the RS; so do models and test suites; and also test logs generated during test execution. This ultimately allows tracking back from test execution results to the specification via the requirements. Requirement coverage is a major way to quantify the quality of a test suite.

2.3. Testing Approach

The PQAP does not prescribe whether to use traditional or model-based testing. Rather, test suite developers are asked to submit a suggestion for the approach after the study phase, which will be considered by reviewers. It is also common to apply a hybrid approach, where model-based testing is used to develop a test suite, but some special scenarios are tested by manually written tests. The hybrid approach is supported by a unique *test adapter* that can be used both from models and from traditional test suites. Test adapters are defined in the managed *Protocol Test Framework* (PTF), an extension of Visual Studio Test Tools. Below, we will first discuss PTF in more detail, and then our particular MBT approach based on Spec Explorer.

3. Test Representation and Test Adapters

One of the major problems for developing test suites for protocols is to get protocol elements on the wire and back. Elements can be data packets in various encodings, remote procedure calls, and so on. This is well known in the protocol testing community, resulting in efforts like TTCN-3 [TTCN3].

3.1. VSTT Extension

For our project, we developed an extension of Visual Studio Test Tools (VSTT), called Protocol Test Framework (PTF), which is based on the Visual Studio Unit Testing framework. Using managed code for writing test adapters has many advantages, among them a high-level programming environment and IDE, languages which are less error prone by definition (we use C#), and access to the .Net framework: Microsoft's major platform for interoperable code. The Unit Testing framework, in addition, provides a concise way to represent test cases and to manage their execution under VSTT.

PTF adds to VSTT custom support for dealing with protocols, including ways to automatically serialize and de-serialize data packets based on declarative definitions in .Net attributes, and access to RPC calls directly from .Net. We use Visual Studio's C++/CLI subsystem, which allows mixed managed/native development, to manage special cases requiring access to the Windows API or other low-level system levels.

3.2. Test Adapters

A central concept of PTF is a *test adapter*. A test adapter defines an interface to the protocol under test at a problem-oriented level of abstraction. It is given as a managed interface that contains methods for sending data to the server, and events for receiving data back. Test adapters are deployed in a service-oriented style. A PTF configuration file associates interfaces with implementations, allowing a test suite to be easily adapted to different environments by just changing the test adapter implementation association in the configuration file.

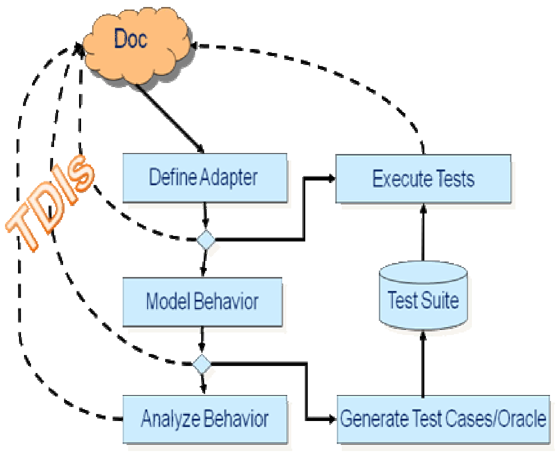
The design of test adapter interfaces is part of the planning phase of the PQAP. Hence, development teams can independently proceed on test suite and adapter implementation during the design phase, based on the adapter interface as a contract.

Test adapters can be implemented in diverse ways in PTF. It is possible to wrap an existing API or program, or to synthesize packets and directly parse them from the wire. The last approach is supported by the Protocol Adapter Compiler (PAC), a tool that extracts definitions of protocol elements directly from the protocol document, and generates data structures which can be consumed by the PTF framework. This is possible since protocol documents have an XML version containing formal tags that specify data packets and remote calls, as well as constraints over them. PAC analyzes this metadata, checks it for consistency, and automatically generates adapters for a large class of protocols from it.

4. Model-Based Test Suite Development

Model-based testing has been discussed extensively in academic and industrial communities (e.g. textbook [9]). The general workflow of how MBT is applied as part of our effort is shown in Figure 2.

Figure 2: MBT in the PQAP



The dotted lines represent feedback towards the document, in the form of so-called TDIs (Technical Document Issues). As seen from Figure 2, TDIs are generated in *all* steps. Typically, more than 50% of the TDIs in our project are actually detected before any test is ever executed, confirming community wisdom that the value of MBT is not only in the testing itself but in the mere development of the model.

4.1. Spec Explorer 2007

Our particular approach to MBT is based on Spec Explorer 2007, the successor of the publically available tool Spec Explorer 2004 [2][3]. This tool has been discussed in [1] and conceptually described in [5]. A text book will soon be released that tackles the general modeling approach [6], and is used in our internal training programs. Here, we summarize the approach.

4.2. Model Programs

Models are written in Spec Explorer 2007 using C# together with a notation called Cord for describing scenarios, test purposes, and model composition rules [7]. C# code is used to represent *abstract state machines* (ASM) [8] by writing methods constituting

guarded update rules over a global state. C# code comprising a model is called a *model program*, in order to distinguish it from normal programs.
The machine defined by a model program is usually not finite (the state space can consist of complex data elements like collections, there can be dynamic object creation, etc.). The process of *exploration* extracts a finite subset of the model program for the purpose of test generation or model-checking.

4.3. Slicing along Test Purposes

Exploration can be controlled by a number of approaches. The one that has proved most successful in our project is based on *slicing* of the model program against a *test purpose* given as an independent model. Slicing results from the composition of a model program with a test purpose. Often, test purposes are defined in a scenario-oriented style, using the language Cord, which allows defining patterns of call sequences with constructs similar to regular expressions.
When designing a model-based test, our test engineers usually begin by defining test purposes in the plan phase of the PQAP. These test purposes can be initially expressed in Cord, or will be later translated into Cord. There can be quite a number of test purposes (in the dozens for a medium sized protocol).
Test purposes are often very abstract. For example, a test purpose for a file server protocol may list a sequence of steps to setup a certain configuration, and after that allow open/creation/read/write/close operations to be intermixed in any order for a bounded number of steps. These “loose” patterns can be naturally described in Cord, mainly because Spec Explorer is based on symbolic exploration. The model program for this example would contain the full model of the file server, which predicts in every state what possible steps and successor states are possible, including potential non-determinism as well as expected outcomes of operations on sets of parameters. An exploration goal is constructed by parallel composition of the test purpose with the model program, whose actual computation by the Spec Explorer tool results in a finite slice of the infinite state space of the model program. This process can be repeated for as many test purposes as desired, using the same model-program.

4.4. Test Generation

Spec Explorer generates test suites from a finite exploration result and stores them in the VSTT Unit Testing format. These test suites directly integrate into the Protocol Test Framework (PTF), and execute against the adapter abstraction, which is fed into the modeling process.

4.5. Model Checking

Spec Explorer also allows for model-checking of behaviors. One goal of model-checking in the context of our project is to verify that certain traces given as samples in the protocol document are actually contained in the modeled behavior; another one is the opposite: to check whether certain invalid traces are *not* contained. Model-checking is thereby performed just as another instance of model-composition, where the properties to check are described as independent models, and composed with the model-program.

4.6. Requirements Tracing

Tracing requirements gathered from the specification all the way to test-run reports is an integral component of our process. Spec Explorer supports associating requirements to preconditions and updates by calling specific library methods in C# code. Requirements covered in each step (and in the path leading to each state) are recorded as part of exploration results and transferred to generated tests. At test-run time, dynamic requirement capturing is logged and output as part of test reports. In addition, PTF provides direct mechanisms for adapter and traditional test code to log manually captured requirements. These combined features provide requirement coverage information all along the process, an essential feedback to drive test-purpose definition, to analyze exploration and model checking results, to debug individual test case execution and to interpret global test suite results.

4.7. Training and Adoption

Without systematic training, test engineers would not be able to ramp up in a relative advanced technology like MBT. We provide training classes to new vendor hires, which run for not more than a few days, and include basic training in modeling, C# and Spec Explorer. The courses are based on the book [6], and taught by the book’s authors. In addition, adoption of the techniques is driven by more experienced test suite developers mentoring less experienced ones.
Also, the reviewing process of the PQAP is an important brick in the adoption progress. Senior experts in the MBT area are assigned as “designated reviewers” to a test suite project, and they also serve as “buddies” for test suite developers, consulting them on a frequent base. As it turns out, one of the biggest bottlenecks of the process is the availability of senior

reviewers. As the project proceeds, we intend to upgrade senior test suite developers to the level of test suite reviewers, for which we have developed a training and certification program as well.

5. Conclusion

There are numerous instances of model based testing being successfully used in the industry. We believe our current effort which intends to apply it for *200 network protocols* will propel MBT to become mainstream in the software industry. The feasibility and scalability of MBT is evident in the fact that for 15 protocols, we have delivered "model to metal" test suites. 20 more protocols are nearing completion.

6. Acknowledgment

This work wouldn't have been possible without the tireless efforts of the PT^3 team, in Redmond, Hyderabad, and Beijing. Thanks also go to colleagues at Microsoft Research which contributed creating the Spec Explorer technology: Colin Campbell, Yuri Gurevich, Lev Nachmanson, Wolfram Schulte, Nikolai Tillmann and Margus Veanes. Special thanks go to the Technical Committee (TheTc [10]) and Harry Saal, as well as the EU trustee Neil Barett and his team, which monitored and influenced this work.

7. References

[1] Wolfgang Grieskamp. *Multi-Paradigmatic Model-Based Testing*. Invited talk in Klaus Havelund and Manuel Nunez and Grigore Rosu and Burkhart Wolff, FATES/RV, LNCS 4262, 2006.

[2] Colin Campbell, Wolfgang Grieskamp, Lev Nachmanson, Wolfram Schulte, Nikolai Tillmann, and Margus Veanes. *Model-based testing of object-oriented reactive systems with Spec Explorer*. Technical Report MSR-TR-2005-59, Microsoft Research, May 2005. To appear in Formal Methods and Testing, LNCS, Springer.

[3] Wolfgang Grieskamp, Yuri Gurevich, Wolfram Schulte, and Margus Veanes. *Generating finite state machines from abstract state machines*. In ISSTA'02, volume 27 of Software Engineering Notes, pages 112–122. ACM, 2002.

[4] Keith Stobie. *Model based testing in practice at Microsoft*. In Proceedings of the Workshop on Model Based Testing (MBT 2004), volume 111 of Electronic Notes in Theoretical Computer Science. Elsevier, 2004.

[5] Wolfgang Grieskamp, Nicolas Kicillof, Nikolai Tillmann. *Action Machines: a Framework for Encoding and Composing Partial Behaviors*. International Journal of Software Engineering and Knowledge Engineering 16(5): 705-726, 2006.

[6] Jonathan Jacky, Margus Veanes, Colin Campbell, Wolfram Schulte. *Model-based software testing and analysis with C#.* Cambridge University Press, 2008 (to appear).

[7] Wolfgang Grieskamp and Nicolas Kicillof. *A schema language for coordinating construction and composition of partial behaviors*. In Proceedings of the 28th International Conference on Software Engineering & Co-Located Workshops – 5th International Workshop on Scenarios and State Machines. ACM, May 2006.

[8] Yuri Gurevich. *Evolving Algebras 1993: Lipari Guide*. In E. Boerger, editor, Specification and Validation Methods, pages 9–36. Oxford University Press, 1995.

[9] Mark Utting and Bruno Legeard. *Practical Model-Based Testing*. Morgan-Kaufmann, 2007.

[10] The Technical Committee. URL: http://www.thetc.org.